On Log Analysis and Stack Trace Use to Improve Program Slicing

Vincent Bushong[1], Jacob Curtis[1], Russell Sanders[1], Mark Du[1], Tomas Cerny[1], Karel Frajtak[2], Pavel Tisnovsky[3], and Dongwan Shin[4]

[1] Baylor University, Waco TX 76706, USA {vincent_bushong1,Russell_-Sanders1,Jacob_Curtis1,Mark_Du1,tomas_cerny}@baylor.edu
[2] Computer Science, FEE, Czech Technical University, Karlovo nam. 13, Prague, 121 35, Czech Republic frajtak@fel.cvut.cz
[3] Red Hat, Brno, Czech Republic ptisnovs@redhat.com
[4] Computer Science, New Mexico Tech, Socorro NM 87801 USA dongwan.shin@nmt.edu

Abstract. Program slicing is a common technique to help reconstruct the path of execution a program has taken. It is beneficial for assisting developers in debugging their programs, but its usefulness depends on the slice accuracy that can be achieved, which is limited by the sources of information used in building the slice. In this paper, we demonstrate that two sources of information, namely program logs, and stack traces, previously used in isolation to build program slices, can be combined to build a program slicer capable of handling more scenarios than either method individually. We also demonstrate a sample application of our proposed slicing approach by showing how our slicer can deduce integer inputs that will recreate the detected error's execution path.

Keywords: Code analysis · Log analysis · Program slicing.

1 Introduction

Finding the root cause of an error is one of the most time consuming and tedious tasks a developer must face. As a result, many methods have been explored to improve the debugging process. One such method depends on program slicing. Program slicing is the process of narrowing down a program's source code to an increasingly-narrow selection. When applied to the problem of debugging, the goal of program slicing is to narrow the code down to the section of code that was actually executed during the erroneous run of the program. This reduces the amount of code that must be analyzed to find the error source, either through manual analysis or through other automated techniques.

While program slices can be constructed using the source code alone, in order to slice a program based on an actual given execution of the program, a source of additional information about the runtime data is needed. One such source of information is the stack trace a program produces when an unhandled runtime exception occurs. The stack trace provides a definite endpoint for the program slice and a cross-section of the functions that were currently executing, so a method based on stack trace analysis already has a good start for building its slice. Another source of data that can be utilized for program slicing is the application log. Logging statements are commonly used in code to provide information about the program's execution by printing errors or other interesting events in a place where developers can read and interpret the messages. Log messages can also be analyzed as part of an automated process, and, if the analyzer has access to the source code that generated them, the log messages can serve as checkpoints in the code. Using the presence of log messages compared to the log statements in the code, a program slicer can more accurately identify which paths were possible for a given execution. Developers already use this technique manually, not knowing it is program slicing; they read log messages and search for its location and meaning in the code to narrow down the problem's location.

Program logs and stack traces have both been used as a source of data to improve the accuracy of program slicing [15,1,10,14,2,13,9,5,14,5]. However, no existing work has combined the two techniques to use simultaneously. In order to narrow down a program slice as much as possible, every available source that can provide information about the execution should be used. We propose a method that uses both the stack trace of an exception and any log messages that have been generated up until the point the exception occurred to create a program slice more accurately than using either data source in isolation.

To demonstrate our method and a potential application for the resulting program slice, we describe an implementation for the Go language. It creates the program slice for a runtime exception and uses it in conjunction with a Satisfiability Modulo Theories (SMT) solver to suggest program inputs that produce the same path through the program as the exception, thus reproducing the error. In section 2, we give our motivation. Section 3 reviews prior work. Sections 4 and 5 explain our proposed method and sample implementation, respectively. We show an example usage in section 5 and offer conclusions in section 6.

2 Motivation

While usually unstructured and designed for human consumption, program logs represent a rich source of information that can be tapped to learn about an application and its operation. When combined with static source code analysis, log analysis techniques can be used to extract insights about a program's performance using log timing information, the presence of anomalies, and error localization within a distributed system. One particular application of log analysis is improving the accuracy of program slicing. If the locations of log statements in the code are matched with logs collected from the application, the program slice representing a particular execution of the application can be more accurately narrowed down.

Other methods have been used to increase the accuracy of program slicing. One method is to use the information contained in stack traces. If a runtime exception occurs and no exception handler is found for it, the stack trace is usually routed to the logging mechanism to prevent the entire program from crashing. If this stack trace is available post-mortem to an analyzer, its contents can be used to narrow down the slice further, based on the functions it contains. This method has been shown to improve the accuracy of a program slice, lending itself to applications such as improving fault localization.

Both log and stack trace analysis can improve program slice accuracy. However, the two techniques have not yet been combined. Since uncaught exceptions are usually printed to the log, having access to stack traces usually means having access to the program logs as well. If the goal is to build as accurate of a program slice as possible without introducing more overhead to the program itself, every possible source of information should be utilized; if access to the program log is available, then there is no need to depend solely on the stack trace to build the program slice, and vice versa. Our goal is to show that these two techniques can be used in conjunction to build a program slice, and the resulting program slice can be used as the basis for further analysis, in this case, providing the user with program inputs that will recreate the execution path that caused the error.

3 Related Work

Automated log analysis is the process of extracting some insights from a program's logs, and it has been used for detecting performance anomalies, fault localization, and identifying security anomalies [3]. A subset of log analysis methods consists of those that map logs to the corresponding location in source code. The general approach is to traverse an Abstract Syntax Tree (AST), creating regular expressions from the logging statements in the source code to match against gathered logs once the program has run. These regular expressions are associated with their location in the code, and when matched with seen logs, give the location in source code, providing partial observation of program execution. This is a common approach [13,15,8,1,2,12,7,6,4], with some variations. For example, in [2], the regular expressions augment to denote that the log occurs inside a loop or conditional branch. In [8], the regular expressions are associated with the class or method they are used in, not the source line itself.

Program slicing is a technique to narrow down a code base to a particular subset, usually a subset that can affect a particular statement, performed by creating a reachability graph between the program's statements [11]. The variant of program slicing we consider is dynamic program slicing. In this method, a program slice is constructed regarding a single execution of the program, resulting in the subset of the program's code that could have affected a particular line of code in a particular execution [11]. When applied to a part of a program known to have failed, this technique helps in debugging and fault localization [11].

Many works [15,1,10,14,2,13,9,5] utilize log analysis to assist with program slicing. The approach we adopt is inspired by Chen et al. [2]. Statements in the code are assigned labels to determine which lines were executed. The labels used in this study are: "may", "must-have", or "must-not", and are assigned on the basis of whether a certain log appeared during execution. The approach accounts for conditional execution and surrounding code having related execution status. If a log appears in a conditional branch, the entire block of statements in that branch is labeled "must-have"; any code in the opposing branch is consequentially labeled as "must-not" have been executed. Similarly, logs that appear in code but not in the output must not have executed and labeled as such. If no logs are present in either branch, it cannot be known which branch was executed without further analysis and is labeled as "may" have executed. A similar approach is taken by [14], with different names for labels being used. The limitation of this method is that the information needed to construct the slice is entirely dependent on the frequency/location of matchable logs within the source code.

Another method used for program slicing is to analyze stack traces to generate method-level execution paths. This is fairly straightforward, as it is known from a complete stack trace a majority of the functions currently executing at the point it is printed. This information gives a good starting point for building a slice. The approach is used by several works that specifically use the information to help localize faults [14,5]. The authors of [9] supplement this call graph generated from the stack trace with information mined from the internet to construct a complete call-graph that could be used to show more program execution information. The drawback of this approach is that the inner workings of each function's execution are not known.

While these two methods do work in isolation, our goal is to show these two sources of information can be used to enhance each others' coverage, more accurately revealing which paths through a program could have been executed. With the stack trace analysis, we gain an overview of program execution that is then supplemented by the more detailed execution trace given by the log analysis. This way, we have a detailed execution path to analyze.

4 Method

Utilizing printed logs matching statements in the source code will let us slice down the program into the paths that must have or may have executed, further allowing us to perform value analysis on the variables along the path. To meet our goal of extracting the executed path and its conditions in a program, our approach takes to input the stack trace produced by an unhandled error, the logs produced from the program before the error occurred, and the source code that produced the error and logs.

The provided stack trace is parsed to find the functions on the stack when the error occurred. This parsed information also provides the root level function for constructing the control-flow graph (CFG), which represents the different execution paths for a given function. This root function has its CFG constructed and is then expanded. To expand the CFG, we find all of the function calls inside of some function and then replace each one with the code inside of its function declaration, meaning we are in-lining the function's contained statements. We also map the arguments passed into the function call to the parameters from the function declaration. This allows us to determine where parameters used in the in-lined statements get their values from. Having the CFG expanded allows us to backtrack from the point of the exception and slice the program based on printed logs matching log templates along the paths we find.

During the backtracking phase of our method, we use the stack trace in conjunction with the provided logs to slice the program's CFG into the paths that must have, may have, or must not have executed. Our log templates are regular expressions created from a log statement in the program. We account for static and dynamic information in the log so that any message produced from that log statement will match the regular expression. Any time a printed log matches a log template from the CFG, that block is labeled as must have executed. Once a block is labeled as must have executed, any other alternative branches (i.e., conditional control flow) that do not have a log statement matching a printed log are then labeled as must not have executed. Any time we encounter a conditional control flow in the CFG, if the preceding block is labeled as must-have executed, both outgoing branches are labeled as may have executed. May have labels can be overwritten to must-have labels if a printed log matching a log template is encountered.

For the sliced execution path, we also extract the conditions for each possible branch combination within the sliced execution path. This includes expressions within a conditional statement and assignments to variables since these are relevant to the SMT solver for determining possible input values. These conditions are recorded during the backtracking phase when each path is being traversed.

Once we have the execution path along with its conditions that need to be analyzed, we use an SMT solver to perform value analysis on the variables found in the path conditions. The only variables reported to the user calling our method are those that are user input (in our method using the heuristic that any variable never assigned or is a program argument is user input). By doing this, we can report to the user the variable values necessary to recreate the erroneous path of execution, which allows us to create a more accurate program slice overall.

5 Implementation

Here we describe an implementation of our method for the Go programming language to demonstrate the effectiveness of our approach.

Stack trace parsing The first step in our process is to analyze the stack trace written to standard output when an exception is unhandled. This stack trace contains the functions that were currently on the stack when the exception occurred, giving a definite list of functions that were at least partially executed and a definite end point for the path (the line where the exception was thrown). Additionally, the stack trace provides the file names that the functions are located in; since the Go language does not compile to an intermediate representation (such as Java's bytecode), the source files themselves are necessary to perform the analysis, as the exact execution path through the code is not recoverable from the executable.

In Go stack traces, lines can contain either a file path and line number or a function call. We used these two types of lines to detect all of the functions that were on the stack and their corresponding file names and line numbers. We then stored the function information into a structure for later use in constructing the CFG and labeling. Once the functions from the stack trace and point of original error are found, we move on to constructing the CFG.

CFG construction Our approach to creating a CFG for some input program is to create a wrapper type around the existing CFG package in the Go standard library. We create wrapper types around the native Go CFG because it only parses on a per-function basis, and our analysis requires a CFG that includes all function calls connected together so we can view an entire execution trace. We created a wrapper interface with two concrete implementations. The function wrapper is the entry point of a function, representing the function signature along with its entry block. The block wrapper wraps the Go CFG package's block representation and has parent and child connections to other wrapper blocks, mimicking the structure of the Go CFG so it can be traversed in a similar way, while also allowing us to store extra information.

We construct the wrapped CFG by first building the Go CFG for the function, then recursively traversing its blocks and creating a block wrapper for each. Every time we create a block wrapper, we replicate the wrapped block's children with a wrapper substitute, allowing us to keep the sequence of the original blocks within our wrappers. The parent relation is also set to the current wrapper on the successor wrapper. As a result, we have a series of blocks where we can traverse through both its ancestors and descendants.

To connect the CFG across functions, we replace function calls with parsed function itself. Beginning with the function wrapper at the root of the stack trace, we search the chain of block wrappers for function calls. Upon finding a call expression, we split the current block in two halves, and search the code for the function declaration matching the call expression. We then construct the called function's function wrapper, passing in the list of arguments found via the call expression in order to map the arguments to the function parameters. The new function is given the first half of the original block as its parent and the second half of the original block as its child. After repeating this for all function calls, a connected CFG for the given stack trace has been created.

Log matching The next step is to find log statements within the source code and construct regular expressions to match them. We extract log statements by examining AST nodes and looking for call expressions that match the log statements' format. In our implementation, we currently detect log calls from the Zerolog library[5] as well as the standard Go logging library and generate a regular expression for the log message that matches both the static part of the message and the dynamic portions. The dynamic portions are variable values that are interspersed in a log message. For example, the dynamic portion in a string "%d is an odd number" would be %d, and the regex generated would include a '\d' to match that portion.

Our log-matching functionality is used later during the labelling phase. Given a set of observed log messages, we attempt to match the messages to each log statement's regular expression. If we identify a match between a log message and a regular expression, we can identify that log statement and the branch it is in as having executed. This gives us better filtering of execution paths that may or may not have occurred.

Path labeling The next step is to attach labels to each of our wrappers specifying whether the block of code it represents must have or may have executed. This is accomplished using both the stack trace and log matching. Our approach labels the tree bottom-up, starting with the block containing the line in the source that caused an exception. From here, as long as there is only one parent block, we label that parent as a must because there is no deviation in execution. A

[5] `https://github.com/rs/zerolog`

block with two parents indicates its parents are part of a conditional statement, which may be arbitrarily nested. We navigate up the tree until all conditional branches have converged, reaching the topmost condition. We then traverse down each branch, labeling every branch as may-have, and stopping when the original block with two parents, mentioned above, is reached. We then resume labeling parents starting with the block above all of the branches just analyzed. If the log matching described above finds a log statement that matches a log seen in the particular execution, its branch is changed to must-have, and any opposing branches are treated as not having been executed.

The next problem we needed to address is a limitation of the Z3 solver. The Z3 solver analyzes assertions as one unit, which means order is ignored, so conditions such as in this set of statements, "`if (x < 0) x = 1; if (x > 0) return`", will contradict. The solver tries to solve a system of equations where `x < 0, x = 1, x > 0`, which is unsolvable. Our solution is to convert the variables in the conditions to a single static assignment form (SSA). SSA treats each reassignment of a variable as if it were a new variable, giving it a unique name. This is necessary because every new value needs to be considered in the conditions. We accomplish this by traversing the tree again and adding numeric values to the beginning of identifiers, incrementing the value for every instance of reassignment seen for a particular variable. Our solution would transform the statements to look like this "`if (x < 0) 1x = 1; if (1x > 0) return`". This allows the solver to assert that `x < 0, 1x = 1, and 1x > 0`, which is solvable.

SMT Analysis After the renaming, we extract the conditions along the execution paths identified by the labeling stage. Following the path of must-have blocks, we add all conditions and assignments of variables to a list, and when necessary, create a new path when a branch is reached to construct every possible route through the code. Any conditions in branches that oppose the branch taken (e.g., an "if" condition that was false when an "else if" condition was true) are negated to provide the most correct set of constraints for the path.

The extracted conditionals (Go AST expressions) along the execution path from the backtracking stage are each passed into a function that recursively converts them into a format that Z3 can recognize. After all of these are converted, they are used as assertions with Z3 to build a model.

If the Z3 solver was able to solve the constraints, we filter the returned value assignments that satisfy the conditions down to only those variables presumed to be user input. We used a heuristic to define user input as any variable that was never given a value assignment in the path conditions. If the argument variable was user input, then the parameter is treated as user input, and vice versa. Command-line arguments were also recognized as user input despite there being an assignment to the variable in the path conditions.

To demonstrate the effectiveness of our method, we show its results of operating on code samples. First example given in Listing 1.1. The listing shows the logs output during the particular execution that would lead to the panic statement on line 24, as well as the simple stack trace generated by the panic. The conditions that are collected include the ones that branch to the locations of collected

Listing 1.1: Example code

```
package ifelse

func main() {
    num, _ := strconv.Atoi(os.Args[1])
    num *= 2
    if num > 4 {
            log.Log().Msgf("%d > 4", num)
    } else {
            log.Log().Msgf("%d <= 4", num)
    }
    x := num * 4
    if x < 9 {
            log.Log().Msgf("%d < 9", num)
    }
    num -= 2
    if num < 0 {
            log.Log().Msgf("%d is negative", num)
    } else if num < 10 {
            log.Warn().Msgf("%d has 1 digit", num)
            panic(errors.New("has 1 digit"))
    } else {
            log.Log().Msgf("%d has multiple digits", num)
    }
}

Stack Trace:
ifelse.main()
 /workspaces/test/test.go:32

Log Output:
2 < = 4
8 < 9
0 has 1 digit

Project root: /workspaces/test/
```

logs in the source code, and the negations of ones that lead to logs that were *not* collected. The following expressions are given to the Z3 Solver after SSA reassignment has been performed: `1main.num == main.num * 2, !(1main.num > 4), main.x == 1main.num * 4, main.x < 8, 2main.num == 1main.num - 2, !(2main.num < 0), and 2main.num < 10`. After these statements are asserted with the Z3 Solver, the variables are filtered to show only variables that are described as user input, as discussed earlier. The final output shows a possible value for the variable 'num' is 1, thus, the expected input given for this example.
Discussion We present two primary contributions. First, combining two previous approaches to program slicing creates a more robust system. The success of any technique that depends on program slicing depends greatly on the accuracy of the slice that can be achieved. Using multiple sources of information increases the likelihood that the slice can be further narrowed down, as each source of information by itself may not contain the full picture. While they will not be applicable in every situation, the program logs and stack trace can be mutually beneficial to each other. Further information sources, such as static configuration files or code from third-party libraries, could be brought into the process to increase the number of scenarios the program slicer can adequately address.

Second, we demonstrate a sample application using our program slicing method. By representing the program slice's path as a series of conditionals,

we were able to obtain a list of constraints that can be programmatically analyzed, in this case, by using an SMT solver. In this case, the output was relatively simple, being input values that could recreate the path taken through the program; there is room for more complex solutions to be built upon this concept, such as recommending ranges of values, as well as considering non-integer values.
Threats to validity As a proof-of-concept prototype work, an in-depth case study has yet to be done to address several threats to the validity of our work. Our prototype's performance on large codebases has yet to be determined, though optimizations may address performance issues to the implementation, rather than the method itself. A case study or series of studies is also necessary to determine how many scenarios the combination of program logs and stack traces affects in the broader corpus of code in the industry.

There a limitation of a CFG in describing modern program execution. While analyzing conditionals as the primary program flow indicator is a good baseline, it leaves out approaches used in emerging software trends, e.g., aspect-oriented programming, interceptors, and distributed programs, all of which defy description by a traditional CFG representation of a piece of code. The new potential execution paths these alternative methods present will have to be analyzed using new tools and techniques; if they could be analyzed and merged into an intermediate representation with the rest of the code flow, common techniques (e.g., our labeling approach using logs and stack traces) could be used on the result.

6 Conclusion

In this paper, we have presented a novel approach to program slicing by combining two sources of information that have previously remained separate, program logs, and stack traces. We have shown that these proven approaches to program slicing are able to be used simultaneously to complement each others' coverage by sharing our implementation of the combined methods. We also presented a sample application of this novel technique in using an SMT solver and the paths of conditional constraints found in the program slice to assist the programmer in recreating the program's execution path.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research https://research.redhat.com.

References

1. Bao, L., Li, Q., Lu, P., Lu, J., Ruan, T., Zhang, K.: Execution anomaly detection in large-scale systems through console log analysis. Journal of Systems and Software **143**, 172 – 186 (2018), `https://doi.org/10.1016/j.jss.2018.05.016`
2. Chen, B., Song, J., Xu, P., Hu, X., Jiang, Z.M.J.: An automated approach to estimating code coverage measures via execution logs. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 305–316 (2018), `https://doi.org/10.1145/3238147.3238214`
3. El-Masri, D., Petrillo, F., Guéhéneuc, Y.G., Hamou-Lhadj, A., Bouziane, A.: A systematic literature review on automated log abstraction techniques. Information and Software Technology **122**, 106276 (2020), `https://doi.org/10.1016/j.infsof.2020.106276`

4. Ghanbari, S., Hashemi, A.B., Amza, C.: Stage-aware anomaly detection through tracking log points. In: Proceedings of the 15th International Middleware Conference. p. 253–264. Middleware '14, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2663165.2663319
5. Jiang, S., Zhang, H., Wang, Q., Zhang, Y.: A Debugging Approach for Java Runtime Exceptions Based on Program Slicing and Stack Traces. In: 2010 10th International Conference on Quality Software. pp. 393–398 (Jul 2010), https://doi.org/10.1109/QSIC.2010.23, iSSN: 2332-662X
6. Pecchia, A., Cinque, M., Carrozza, G., Cotroneo, D.: Industry practices and event logging: Assessment of a critical software development process. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2. pp. 169–178. ICSE '15, IEEE Press (2015), https://dl.acm.org/doi/abs/10.5555/2819009.2819035
7. Schipper, D., Aniche, M., van Deursen, A.: Tracing back log data to its log statement: From research to practice. In: Proceedings of the 16th International Conference on Mining Software Repositories. pp. 545–549. MSR '19, IEEE Press (2019), https://doi.org/10.1109/MSR.2019.00081
8. Shang, W.: Bridging the divide between software developers and operators using logs. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 1583–1586 (2012), https://dl.acm.org/doi/10.5555/2337223.2337490
9. Sui, L., Dietrich, J., Tahir, A.: On the use of mined stack traces to improve the soundness of statically constructed call graphs. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). pp. 672–676 (2017), https://ieeexplore.ieee.org/document/8306000
10. Sun, C., Ran, Y., Zheng, C., Liu, H., Towey, D., Zhang, X.: Fault localisation for ws-bpel programs based on predicate switching and program slicing. Journal of Systems and Software **135**, 191 – 204 (2018), https://doi.org/10.1016/j.jss.2017.10.030
11. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (Mar 2005), https://doi.org/10.1145/1050849.1050865
12. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 117–132. SOSP '09, Association for Computing Machinery, New York, NY, USA (2009), https://doi.org/10.1145/1629575.1629587
13. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S.: Sherlog: Error diagnosis by connecting clues from run-time logs. In: Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 143–154. ASPLOS XV, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1736020.1736038
14. Zhang, H., Jiang, S., Rong Jin: An improved static program slicing algorithm using stack trace. In: 2011 IEEE 2nd International Conference on Software Engineering and Service Science. pp. 563–567 (Jul 2011), https://doi.org/10.1109/ICSESS.2011.5982378, iSSN: 2327-0594
15. Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D., Stumm, M.: Lprof: A non-intrusive request flow profiler for distributed systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. p. 629–644. OSDI'14, USENIX Association, USA (2014), https://dl.acm.org/doi/10.5555/2685048.2685099